ASSISTANT COMMISSIONER OF PATENTS
Washington, D.C. 20231

DOCKET NUMBER: UK998094

Sir:
Transmitted herewith for filing is the Patent Application of:

Inventor(s): **E K Kolodner and M J Trotter**

For: **VIRTUAL MACHINE MEMORY MANAGEMENT**

Enclosed are:

X Patent Specification and Declaration

X 6 sheets of drawing(s)

X An assignment of the invention to International Business Machines Corporation (includes Recordation Form Cover Sheet)

X A certified copy of a United Kingdom application filed on 23 December 1998, Serial Number 9828298.1

___ Information Disclosure Statement, PTO 1449 and copies of references

The fee has been calculated as shown below:

For	Number Filed	Number Extra	Rate		Fee $
Basic Fee					$ 760.00
Total Claims	27 - 20	7	x 18	=	126.00
Independent Claims	3 - 3		x 78	=	
MULTIPLE DEPENDENT CLAIM PRESENTED			260	=	
			TOTAL		$ 886.00

X Please charge my Deposit Account No. 09-0468 in the amount of $886.00. A duplicate copy of this sheet is enclosed.

X The Commissioner is hereby authorized to charge payment of the following fees associated with this communication or credit any overpayment to Deposit Account 09-0468 . A duplicate copy of this sheet is enclosed.

X Any additional filing fees required under 37 CFR §1.16.

X Any patent application processing fees under 37 CFR §1.17.

CERTIFICATE OF MAILING

I hereby certify that the above paper/fee is being deposited with the United States Postal Service as first class mail in an envelope addressed to the Commissioner of Patents and Trademarks, Washington, DC 20231

Date of Deposit June 22, 1999

Person mailing paper/fee Wendy R. Petrovich

Signature ______________________

Respectfully submitted

Jay P. Sbrollini
Registration No. 36,266
IBM Corp, IP Law Dept,
T J Watson Research Center, PO Box 218,
Yorktown Heights, New York 10598.
Telephone: (914)945-2587

APPLICATION

FOR

UNITED STATES LETTERS PATENT

INTERNATIONAL BUSINESS MACHINES CORPORATION

VIRTUAL MACHINE MEMORY MANAGEMENT

FIELD OF INVENTION

This invention relates to memory management in a multithreaded runtime environment and in particular to garbage collecting of storage objects with respect to a local stack.

BACKGROUND OF INVENTION

The Java programming language has its origins in a project undertaken by Sun Microsystems to develop a robust programming environment that would meet the technical challenges of the consumer device software environment. The original consumer device projects were eventually abandoned but the Java programming language found itself being used on the World Wide Web to enable cross platform operation of programs downloaded from the internet. It is simple to use having similar features to C++ such as the basic object orientated technology but without some of the more complex features.

Typically, Java applications (source code) are compiled by the Java compiler into Java byte code (intermediary code or pseudo object code) which can be loaded and executed by a Java Virtual Machine (JVM) (see Figure 1). The JVM provides an instruction set, memory management and program loading capability that is independent of the hardware platform on which it has been implemented. The Java application source code is compiled into architecture independent byte code and the byte code is interpreted by a JVM on the target platform. Java is designed to be portable and follows some defined

portability standards, which intend the source code to be "write once, run anywhere". The Java byte code may be further compiled into machine code (object code) for the target platform at which point the architectural independent nature of Java is lost.

The JVM is a software computing machine, effectively it simulates a hardware machine that processes Java byte code. The byte code is interpreted and processed by a JVM such as an Windows JVM running on a Intel personal computer platform. The JVM includes components for loading class files, interpreting the byte code, garbage collecting redundant objects, and for managing multiple processing threads. The JVM may also include a Just-In-Time compiler to transform some or all the byte code into native machine code.

Multithreading is a feature built into the Java language to allow users to improve interactive performance by allowing operations to be performed while continuing to process user actions. Multithreading is similar to multitasking, but whereas multitasking allows many applications to run on the same system in several processes, multithreading allows many routines (threads) in one application to potentially run in parallel within one process.

Garbage collection is the term used for describing how program objects are automatically discarded by the system after they have been loaded into memory and after they are no longer useful.

For further information on garbage collection see Chapter 1 of 'Garbage Collection' by H Jones & R Lins, Wiley. Chapter 4 deals with Mark & Sweep techniques.

Many current implementations of Java use the classic mark-sweep-compact method of garbage collection as delivered in the base SUN JVM. References to the objects that are being processed at any instant by the system are stored in the registers, one or more thread stacks and some global variables. The totality of objects that may be needed by the system can be found by tracing through the objects directly referenced in the registers, stacks, and global variables and then tracing through these "root" objects for further references. The objects in use by a system thereby form a graph and any extraneous objects are not part of this graph. Once all the objects in the graph are found, the remaining objects in the heaps may be discarded (garbage collected).

The traditional mark and sweep garbage collection method is described below in terms of pseudo code with respect to a single heap:

- Stop all threads causing the active registers for each thread to be stored in its stack
- Trace all stacks for object references - the local roots
- Trace all global variables for object references - the global roots
- Trace through root set for references until no new object references (the sum of the local and global roots is the root set).
- Delete all objects in the single heap that are not referenced

There are problems with this technique in a multi-threaded and long running environment. The first problem is that in order to garbage collect all the

threads must be stopped in order to work out what objects are unreachable (there are no pointers to them in the global or local variables and no pointers to them in other reachable objects. Various authors have attempted to solve this problem. One approach is an on-the-fly collector which does not stop all threads, however it cannot compact the reachable objects leading to fragmentation. Another approach are the generational scavenging schemes, which reduce the size of the set of traced objects by concentrating effort on the most recently allocated objects; however, these schemes must stop all of the threads. In an ideal world we would like to achieve a collector which works independently on all threads and compacts the local heap of the threads to maximise the free space available.

Another solution attempts to achieve this in a language (ML) other than Java by taking advantage of immutable objects which can be placed in thread-local heaps. An immutable object is non-modifiable and when such an object become reachable globally a copy of the object can be made in the global heap. Clearly this technique is only applicable to languages defining immutable objects.

Another approach moves an object into the global heap on first use. The difficulty here is that in order to move the object, references from elsewhere to it must be updated; in an environment where objects are referenced by handles this is made easier although there are still cases where objects cannot be moved. Unfortunately handles bring their own problems and the IBM ports of the JVM have removed handles to improve

performance and remove the need to subdivide the heap into handles and object spaces.

SUMMARY OF INVENTION

According to one aspect of the invention there is provided a method of managing memory in a multi-threaded processing environment including respective local thread stacks and heaps and a global heap, said method comprising: creating an object in a thread heap; and monitoring whether the object is reachable from other than the registers or stack of the thread which created it.

Preferably the method further comprises: associating a local status with the object; and changing the status of the object to global under certain conditions.

More preferably the method further comprises deleting from the thread heap one or more local objects when they are not reachable from a local root.

Advantageously where reachability is determined by tracing from the local root.

More advantageously the status of an object in the thread heap is changed to global if the object is assigned to a static variable or if the object is assigned to a field in any object. This includes objects in the thread heap, any other thread heap or the global heap.

Hence this proposal concentrates on the case of implementing a stack local heap which has not been done before. In essence the solution is to keep track of object references using write barriers placed in any

operation which assigns references eg putfield, putstatic and aastore.

The 'putfield' operation for example is a java bytecode which causes the value on the top of the stack to be placed in the object reference below it on the stack. The object field to be updated is defined by the constant pool reference (2 bytes) which follows the bytecode. See p.325 'Java Virtual Machine Specification', Lindholm & Yellin 1997.

A flag is associated with each object in the heap. In practice the flag is easily provided by ensuring that all objects on the heap are allocated on an 8 byte boundary ensuring that the low three bits are 0. Two of these bits are currently used to denote pinned or free objects leaving a spare flag which we use to denote 'global'. When an object is allocated it is generally placed in a piece of storage associated with the allocating thread; the thread local heap. The global flag is unset at this point. From now on the write barriers implement the following rule: if a reference to a non-global object is assigned to any other object then the referenced object also becomes global. That is that object is no longer a part of the stack local objects

When an object is no longer stack-local it needs to be marked by setting a flag in the object. When the flag is set there is the option of moving the object to another heap of 'referenced' objects. This move can be done very cheaply since an accurate scan of only the stacks need be made looking for references to the previous object location and update them to the new location. By definition the operation can only affect

one object since if the object is referenced by others then it will have already been made 'referenced'.

Hence the move will be very simple. Note that there have been previous schemes in which objects are moved on assignment but in general they depend for their implementation on indirect referencing of objects (handles). This allows a single update following movement of the object.

By keeping track of the set of objects which are only referenced from the stack there may be implemented an efficient move in the absence of handles. Note that the set of objects reachable from the stack can always be compacted very cheaply since the contents of those objects do not need to be examined during compaction; any objects which are referenced cannot be part of the stack local set. This can lead to highly efficient compaction of this restricted set of objects. The contention is that this will reduce the frequency of garbage collection by ensuring that there is always space available to allocate the 'transient' objects which are so prevalent in Java. These can be allocated into a totally compacted heap (zeroth generation or Eden) in which the allocator is a trivial pointer movement and can be implemented inline. Movement out of this heap may optionally be done as some of the new objects are referenced from other objects. This automatically 'smears out' the garbage collection operation to reduce the pausing which is the key problem in eg interactive applications.

Clearly there needs to be type accurate tracing of the stacks to gain the full benefit of this approach but even with a conservative scheme (forcing marking rather

than moving the objects) there would be a benefit of a very simple mark/sweep to free up space in the local heap. It is worth noting that the write barrier required for this technique is significantly simpler than that for a full thread local scheme in that the current 'locality' of the object is unimportant; thus the barrier becomes an unconditional flag assignment. With this simple scheme we would effectively be reducing the frequency at which a global garbage collection is required and thus performance will increase providing that the stack-local set is significant for the given application. A full thread local scheme implements the following rules:

if an object reference is assigned to a static variable the object becomes global; and

if a reference to a non-global object is assigned then the referenced object becomes global.

Thus at any point the thread local heap will contain objects which have the global flag unset and some which have the flag set. Any objects with the flag unset can be garbage collected by the thread itself. The objects marked global must at this stage be treated as reachable, pinned objects; they must not be moved or deleted. Any garbage collection policy can be adopted including compaction for the objects in the local heap, the only constraint is that we must of course compact into the potentially fragmented spaces left between the global objects. Note that the write barriers described above are extremely simple and can be implemented to have a very small effect on performance.

At some point the global objects will be moved from the local heaps into the global heap and this must be done using the current collection approach of stopping all threads. The belief is of course that the frequency of such events can be dramatically reduced and the work to be done also reduced. The global objects need to be removed and put in their own heap, once that is done the individual threads can be left to continue any tidying they need to do on their own heaps. The great improvement is that the moving of global objects is the only part which must be done whilst all program threads are stopped. Tidying of the local heaps can be then done in parallel. The current approach forces the compaction of the entire heap (moving objects) to be done by one thread and all others must wait for completion.

BRIEF DESCRIPTION OF DRAWINGS

In order to promote a fuller understanding of this and other aspects of the present invention, an embodiment will now be described, by way of example only, with reference to the accompanying drawings in which:

Figure 1 is a schematic representation of a platform supporting the embodiment of the present invention;

Figure 2 is a schematic representation of a Java Virtual Machine embodying the invention;

Figure 3 shows the structure of objects in a thread heap;

Figure 4 is a flow diagram depicting the memory management process of the invention;

Figure 5 is a flow diagram of the process of the write barrier; and

Figure 6 is a schematic representation of objects in a thread heap and the global heap of the JVM.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENT

Referring to Figure 1 there is shown a computer platform 10 such as pentium processor based PC typically comprising motherboard with processor, 32Mbytes RAM memory, and interface circuitry; 4G Byte hard drive; 20x CD ROM; SVGA monitor; keyboard and mouse. On power on the platform loads into the memory an operating system 12 such as Windows NT v4 and subsequently a Java Virtual Machine (JVM) based on the Java Development Kit (JDK) v1.1 or v1.2. The Java virtual machine is capable of loading a Java application 16 from disk into memory so that the application may be executed on the platform. The Java application 16 is object orientated program code and creates Java language objects 18A, 18B, 18C, 18D. An example of how the objects are built by the JVM is described with reference to Figure 3.

The JVM 14 comprises code components for the multitude of tasks it has to perform (see Figure 2). The components of relevance to the present embodiment comprise memory manager 20, Java interpreter 22 and just-in-time compiler 24. In operation the JVM creates memory space, the runtime data area 26, for storing the objects 18 and data for the application 16.

The runtime data area 26 comprises thread memory space 28A, 28B, 28n, global heap 34, class area 36 and compiled method area 38. The thread memory space 28 ideally stores local thread objects, local thread execution data and also objects marked global. Local

objects are objects which are used by one thread alone and are not common between. Thread objects are stored in thread heaps 32A, 32B, 32n and thread execution data including object references and register data are stored in the thread stacks 30A, 30B, 30n. The global heap 34 stores objects which are common to more than one thread, that is global objects. A local object which is loaded into a thread heap 32A for processing by thread 28A may become global if it is later used by another thread, say thread 28B, and may remain in the thread heap 28A or be moved to the global heap 34 by 'stop everything'. Class area 36 stores the classes as they are loaded into the JVM, classes are defined as global because they are common to all the threads hence any field of a class is globally reachable. An object which is an instance of class, is placed in a thread heap and is initially local because it may only be used by that thread. An exception is instance objects of class Thread or Runnable which are global at creation because they can be reached by other threads. Whereas an object referenced by a variable in a class is global because the object is reachable from the class which is reachable by all threads. The compiled method area 38 is used by the JIT compiler 24 to store native executable code compiled from the Java byte code.

The memory manager 20 includes an object allocator 21A and a garbage collector 21B which between them control the creation and deletion of objects within the thread heaps 32A, 32B, 32n and the global heap 34.

The Java interpreter 22 includes functionality to process the multitude of Java byte codes which make up the Java byte code function set. A write operation

component 23A processes java write operations such as 'putfield', 'putstatic', 'aastore' using the operating system and microprocessor of the platform. Putfield and aastore are write operations which among other things assign a source object reference to a field in a target object. This means that the target object will contain a reference (or pointer) to the source object; the target object is the object that will be updated by the putfield operation. Putstatic assigns an object reference to a static variable in a class. A write barrier component 23B checks the status of the target object and assigns a local or global status to the source object depending on a certain rule: if a non-global source object is assigned to a reference in any target object then the referenced source object becomes global.

The Java write operation 'putfield' may set a field within a target object to point at a source object using the operating system commands or platform operations . A 'putfield' write operation 23A together with the write barrier 23B is described below in pseudo platform code with comments.

If field to be updated in target object is of 'reference' type.

```
if (source is not global)      {If so check to see
                             if the source has already
                             been set global
set source to global         {If not already set then
                             set source to global
set slot (target, field number, source)
```

{sets a field variable within a target object to point at a source object.

The last instruction code is the write operation 23A whereas the instructions before that constitute the write barrier 23B. Similar pseudo code may be provided for the putstatic and aastore java commands, and other JVM actions, which store references in objects.

The JIT compiler 24 is similar to the Java interpreter 22 in that it contains a write operation component 25A and a write barrier component 25A. However instead of interpreting java byte code in the methods of objects and processing the operations in real time, native code similar to the pseudo code above is created and stored in the compiled method area 38 for execution directly by the platform and operating system whenever that particular method is invoked.

Objects 18 are stored in the thread heaps 32 using multiples of 8 bytes (see Figure 3). Each object has a length word attached which identifies the length of the object in bytes and allows objects in the heaps to be scanned sequentially from the start of the heap. The length word (4 bytes long in the embodiment) has three spare bits available because of the 8 byte alignment boundary and one of these bits is used as a flag for the object to store the local or global status. In the embodiment shown in Figure 3 each of the objects is 40 bytes long which is 5*8 bytes or '00101000' bytes in binary; the last three '000' being spare. This is indicated at the high end of the word in each case. The

low end of the word is spare and thus the first bit of the length word is used to indicate the local/global status, in this case object 18A and 18B are set to '1' to indicate that they are global and 18C is set to '0' to indicate that it is local.

The process of memory management is described with reference to Figure 4. Step 4.1 an object is created from a class stored in the class area 36. Step 4.2 check to see if the class is a global class, ie a class all of whose instances are global or one whose instances we expect to quickly become global, whereby the object is assigned global and/or placed in the global heap (Step 4.7). Step 4.3 the size of the object is calculated to see whether it will fit in the thread heap. Step 4.4 if there is not enough space then garbage collection is performed to free up memory. Step 4.5 place the object in the thread heap memory along with the object length in multiples of eight. Step 4.6 Use a spare bit in the length attribute as a flag and set as local ('0'). The process ends at step 4.8.

The write operation and write barrier process are described with reference to Figure 5. Step 5.1 a write operation is called and intercepted by the write barrier which is integrated with the write operation code. Step 5.2 A check is made as to whether the object is being assigned to any other object and if so set the status of the object as global (step 5.3). Once complete continue with the write operation (step 5.4).

The garbage collector 21A for single thread garbage collection is described in terms of pseudo code with comments.

Trace stack in thread	{Identify objects in use by the thread - these are held in the thread stack
Delete unused objects	{those objects in the heap which are not identified in trace or are not identified as global
Check heap size	{If not enough memory then optionally increase size of the local heap. If still not desirable or possible then do 'full' garbage collection

Full garbage collection is traditional Mark/Sweep performed with all other threads stopped. During full garbage collection global objects are moved from local to the global heap if possible. Object movement is not possible if conservative tracing is performed and the object is referenced directly from the stack because of the uncertainty of the tracing.

The Java language pseudo code program below (together with comments) creates objects representing a bike with two wheels and a bell in a local thread which is put into a global shed. The bike and wheels are make global whereas the bell remains local. If a further new bike object were to be created and the thread heap had no

more memory then the garbage collector would mark and sweep the bell object so long as it was not pointed to by the stack.

```
// Demonstrate the Stack local heap in action.
// For simplicity all niceties such as constructors have
been omitted and the example made deliberately unreal to
illustrate the basic point.  Objects generally start life
local and become global by assignment.

class Shed {    // This is a fake class just to illustrate
                a static variable in operation.
  static Bike bike;
                // The main method will run the sample
                code to illustrate creation of local
                objects which later will become global.
  public static void main(String args[]) {
    Bike  b = new Bike();       // build a new Bike object.
                                The Bike class is loaded
                                and initialised and an
                                instance of Bike is created
                                in the local heap.  It is
                                marked 'local'.
    Bell  c = new Bell();       // build a new Bell object
                                The Bell class is loaded
                                and initialised and an
                                instance of Bell is created
                                in the local heap.  It is
                                marked 'local'.
    Wheel w = new Wheel();      // build a new wheel
                                object. The Wheel class is
```

```
                                    loaded and initialised and
                                    an instance of Wheel is
                                    created in the local heap.
                                    It is marked 'local'.
    b.wheel1 = w;                   // Assign the first wheel
                                    of the bike Java bytecode
                                    putfield, write barrier
                                    detects that w is being
                                    assigned to a field of b
                                    and hence b becomes global.
    Shed.bike = b;              // Assign b to the static field
                                in bike Shed. Java bytecode
                                putstatic is used and thus b is
                                marked global.  Since w is
                                reachable from b then w must be
                                marked global, it already is.
    w = new Wheel();                // build another new wheel
                                    object Another instance of
                                    Wheel is created in the
                                    local heap.  It is marked
                                    'local'.
    b.wheel2 = w;                   // Assign the second wheel
                                    of the bike Java bytecode
                                    putfield, write barrier
                                    detects the assignment and
                                    hence w is marked global.
   // Note that at this point we have built a bike, two
wheels and a bell.  The bell is the only object which is
still local since the bike has been assigned to the
static variable 'bike'.
  }
```

```
}
class Bike {
  Wheel wheel1;
  Wheel wheel2;
  Bell  bell1;
  String name;
}
class Wheel {
  int diameter;
}
class Bell {
  int volume;
}
```

In summary there is described a method and system for memory management in a virtual machine or operating system and in particular an object creation and garbage collection method and system. There is described a method and system of managing memory in a multi-threaded processing environment such as a java virtual machine comprising: creating an object in a thread heap; associating a status with the object and setting the status as local; using write barriers to change the status to global if the object is assigned to a static variable or if the object is assigned to a field in any other object; and performing garbage collection by deleting from the thread heap, when memory space in the thread heap is required, one or more local objects which can not be traced to the thread stack.

The platform in the embodiment does not need to be pentium based nor is restricted to the hardware stated. Any platform which is capable of supporting JDK v1.1 or v1.2 would be capable of supporting the virtual machine of embodiment. Although the embodiment is described with reference to a Java Virtual Machine it is not necessarily so restricted to a virtual machine but may be used in any environment where storage is garbage collected to free up memory in that environment. For example, such an environment could be an multithreaded Lisp runtime or the runtime for some other multithreaded garbage collected language.

Although we have described in the preferred embodiment the local/global status as a field in the object it may be implemented in other ways for instance as a separate table.

Java and Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc.

Microsoft, Windows, and Windows NT are trademarks of Microsoft Corporation.

Now that the invention has been described by way of a preferred embodiment, various modifications and improvements will occur to those person skilled in the art. Therefore it should be understood that the preferred embodiment has been provided as an example and not as a limitation.

CLAIMS

1. A method of managing memory in a multi-threaded processing environment including respective local thread stacks and heaps and a global heap, said method comprising:

creating an object in a thread heap; and

monitoring whether the object is a local root.

2. A method as claimed in claim 1 further comprising:

associating a local status with the object;

changing the status of the object to global under certain conditions.

3. A method as claimed in claim 2 further comprising deleting from the thread heap one or more local objects when they are not reachable from a local root.

4. A method as claimed in claim 3 where reachability is determined by tracing from the local root.

5. A method as claimed in claim 4 wherein the status of an object in the thread heap is changed to global if the object is assigned to a static variable or if the object is assigned to a field in any other object.

6. A method as claimed in claim 3 further comprising intercepting assignment operations to an object in a thread heap to assess whether the object status should be changed.

7. A method as claimed in claim 6 wherein the multithreaded processing environment is a virtual machine.

8. A method as claimed in claim 7 wherein the virtual machine comprises an interpreter and the write operation code in the interpreter is modified to perform the checking of assignment of the object.

9. A method as claimed in claim 8 wherein the virtual machine comprises a just in time compiler wherein native compiled write operation code includes native code to perform the checking of assignment of the object.

10. A method as claimed in claim 9 further comprising using spare capacity in the object header for the flag.

11. A method as claimed in claim 10 further comprising using multiples of 2 or more bytes in a thread heap to store the objects whereby there is at least one spare bit in the object length variable and using the at least one spare bit as the flag.

12. A method as claimed in claim 11 further comprising moving objects whose status is global from the thread heap to a global heap.

13. A method as claimed in claim 12 further comprising compacting the reachable local objects in a thread heap.

14. A method as claimed in claim 1 wherein certain objects are associated with a global status on creation.

15. A method as claimed in claim 14 where said certain objects include Class objects, Thread objects and Runnable objects.

16. A method as claimed in claim 14 further comprising the step of analysing whether an object is likely to be made global and associating such an object with a global status on creation.

17. A method as claimed in claim 16 further comprising allocating objects assigned as global on creation to the global heap.

18. A system for managing memory in a multi-threaded processing environment comprising:

respective local thread stacks and heaps;

a global heap;

means for creating an object in a thread heap; and

means for monitoring whether the object is a local root.

19. A system as claimed in claim 18 further comprising means for associating a local status with the object and means for changing the status of the object to global under certain conditions.

20. A system as claimed in claim 19 further comprising means for deleting from the thread heap one or more local objects when they are not reachable from a local root.

21. A system as claimed in claim 20 further comprising:
means for changing the status of an object in the thread heap to global if the object is assigned to a static variable or if the object is assigned to a field in any other object.

22. A computer program product stored on a computer readable storage medium for, when executed on a computer, managing memory in a multi-threaded processing environment including respective local thread stacks and heaps and a global heap, said product comprising:
means for creating an object in a thread heap; and
means for monitoring whether the object is a local root.

23. A product as claimed in claim 22 further comprising:
means for associating a local status with the object;
means for changing the status of the object to global under certain conditions.

24. A product as claimed in claim 23 further comprising means for deleting from the thread heap one or more local objects when they are not a local root.

25. A product as claimed in claim 24 where reachability is determined by tracing from the local root.

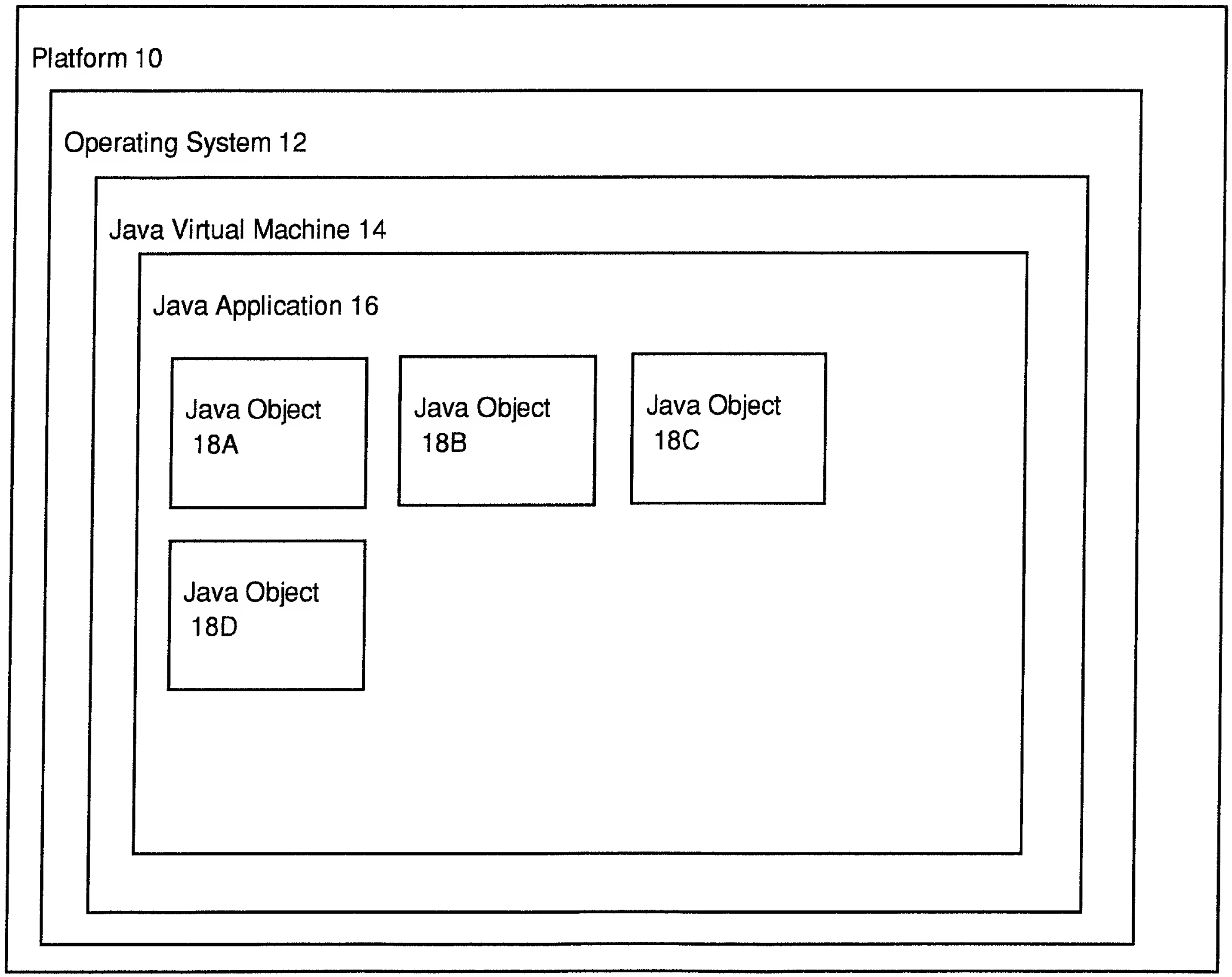
26. A product as claimed in claim 25 wherein the status of an object in the thread heap is changed to global if the object is assigned to a static variable or if the object is assigned to a field in any other object.

27. A method as claimed in claim 4 wherein the status of an object in the thread heap is changed to global if the object is assigned to a static variable or if the object is assigned to a field in a global object.

ABSTRACT

VIRTUAL MACHINE MEMORY MANAGEMENT

This invention relates to memory management in a virtual machine or operating system and in particular to object creation and garbage collection. There is described a method and system of managing memory in a multi-threaded processing environment such as a java virtual machine comprising: creating an object in a thread heap; associating a status with the object and setting the status as local; using write barriers to change the status to global if the object is assigned to a static variable or if the object is assigned to a field in any other object; and performing garbage collection by deleting from the thread heap, when memory space in the thread heap is required, one or more local objects which can not be traced to the thread stack.



Platform 10
Operating System 12
Java Virtual Machine 14
Java Application 16
Java Object 18A
Java Object 18B
Java Object 18C
Java Object 18D

Figure 1

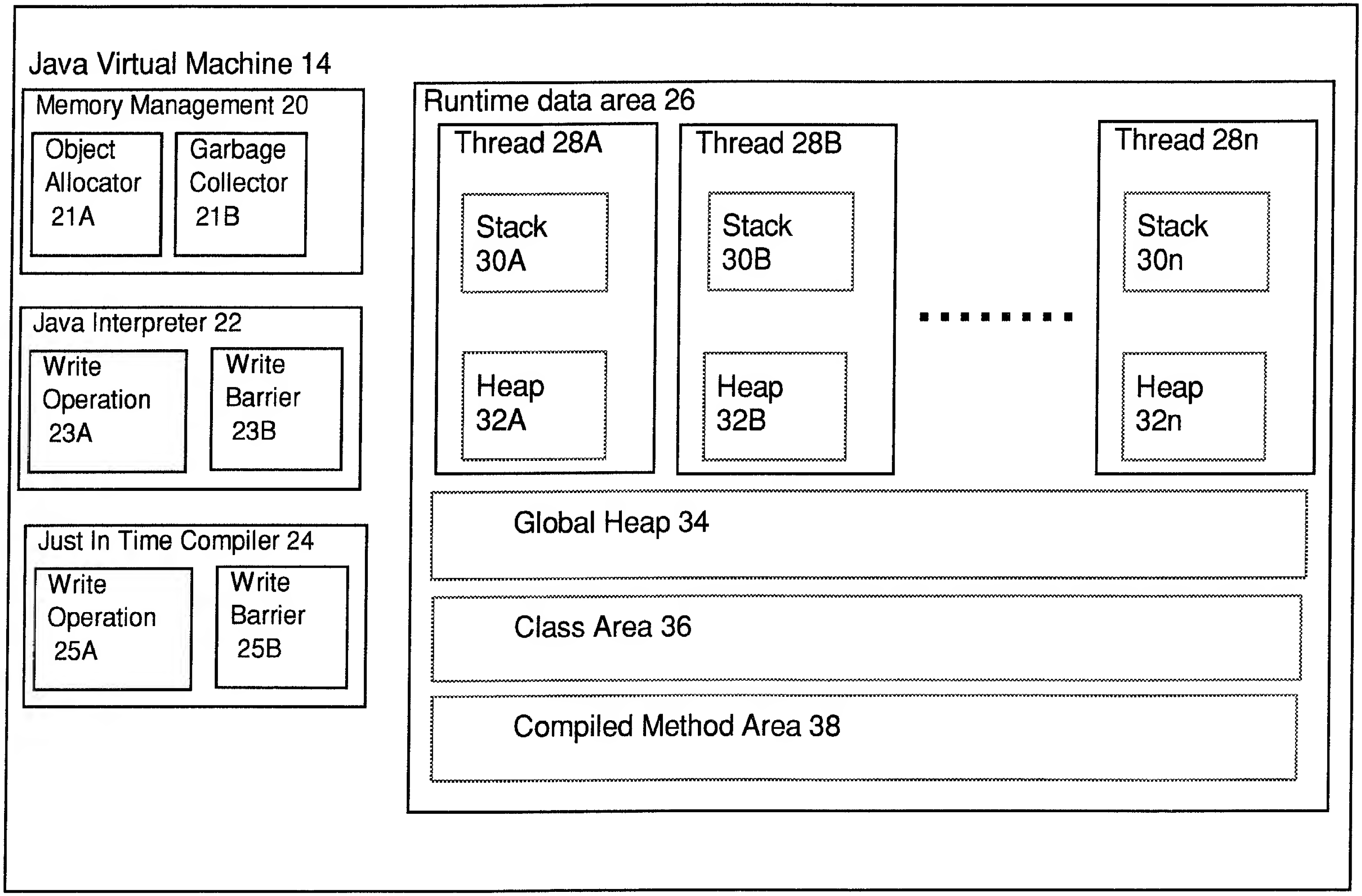
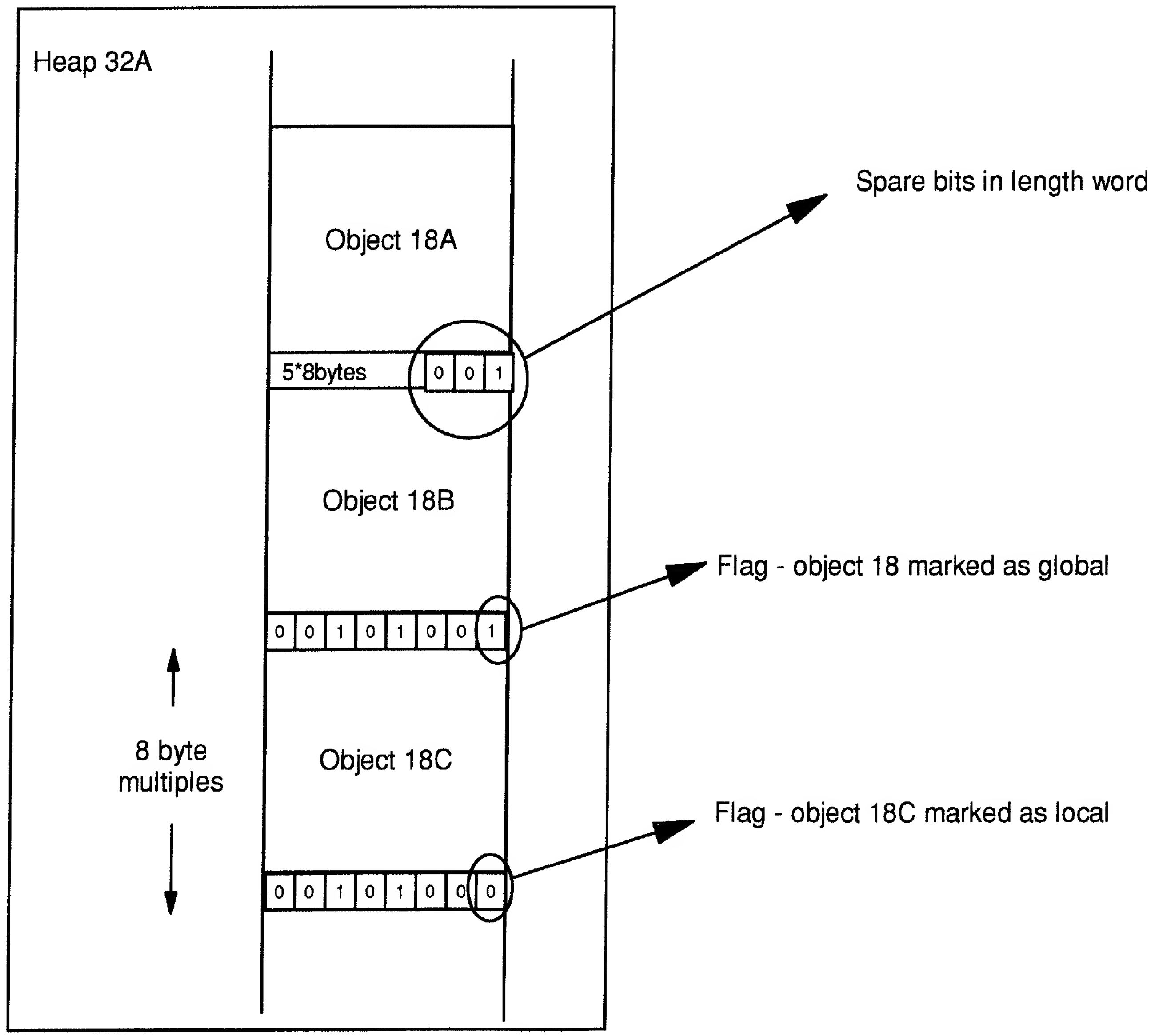

Java Virtual Machine 14
Memory Management 20
Object Allocator 21A
Garbage Collector 21B
Java Interpreter 22
Write Operation 23A
Write Barrier 23B
Just In Time Compiler 24
Write Operation 25A
Write Barrier 25B
Runtime data area 26
Thread 28A
Stack 30A
Heap 32A
Thread 28B
Stack 30B
Heap 32B
Thread 28n
Stack 30n
Heap 32n
Global Heap 34
Class Area 36
Compiled Method Area 38

Figure 2



Heap 32A
Object 18A
5*8bytes
0 0 1
Spare bits in length word
Object 18B
0 0 1 0 1 0 0 1
Flag - object 18 marked as global
8 byte multiples
Object 18C
0 0 1 0 1 0 0 0
Flag - object 18C marked as local

Figure 3

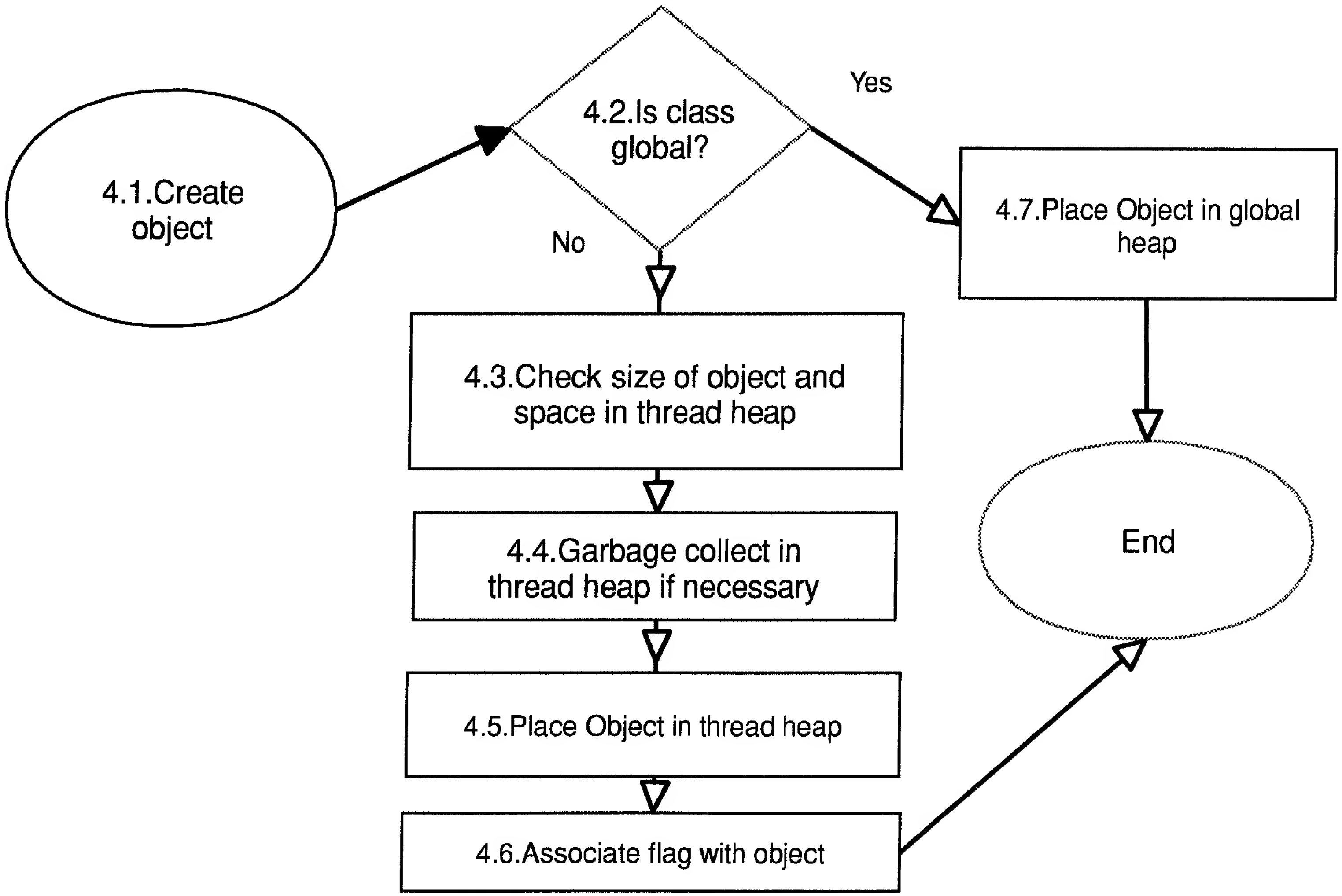

4.1.Create object
4.2.Is class global?
Yes
No
4.3.Check size of object and space in thread heap
4.4.Garbage collect in thread heap if necessary
4.5.Place Object in thread heap
4.6.Associate flag with object
4.7.Place Object in global heap
End

Figure 4

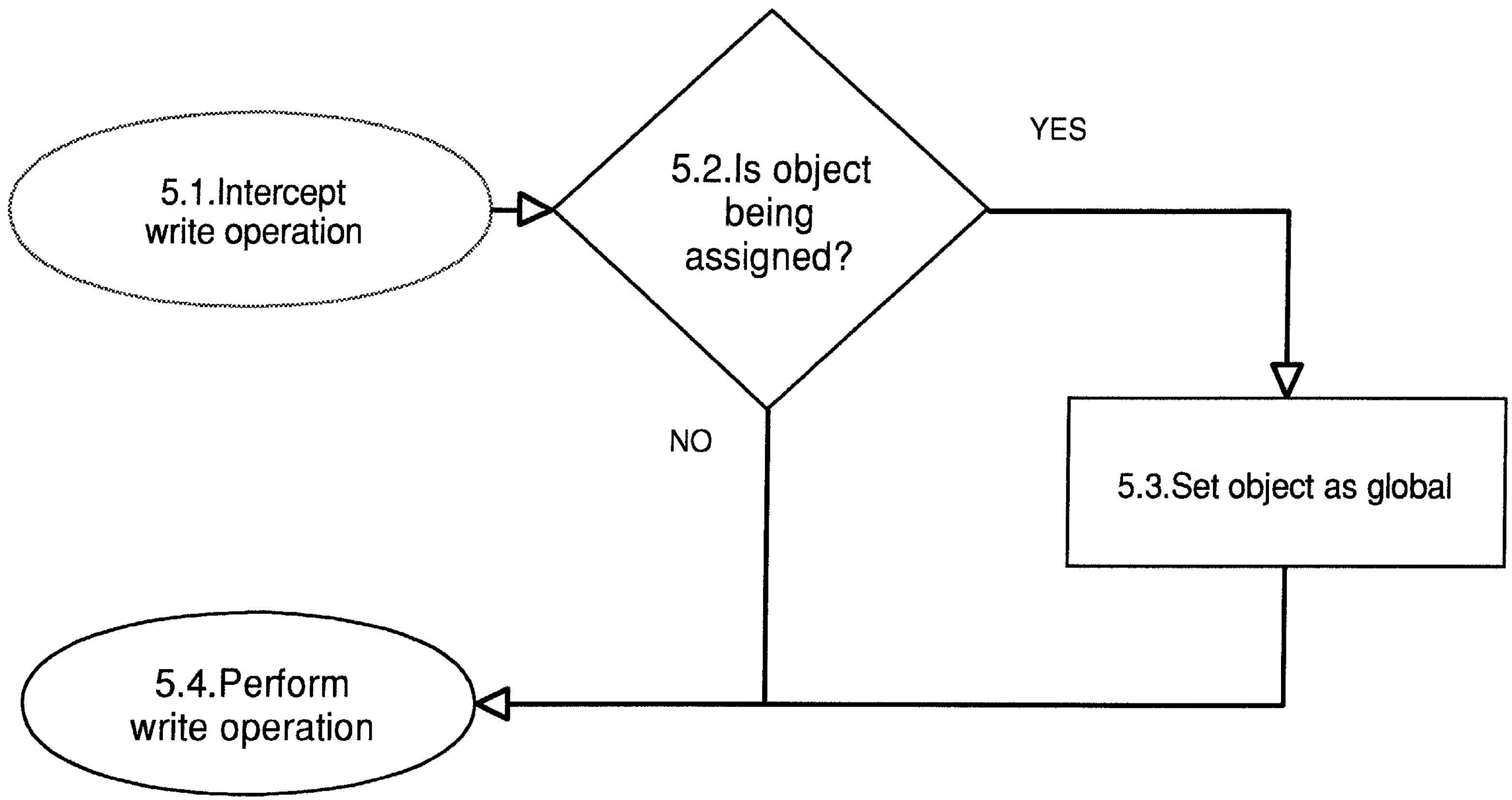

5.1.Intercept write operation
5.2.Is object being assigned?
YES
NO
5.3.Set object as global
5.4.Perform write operation

Figure 5

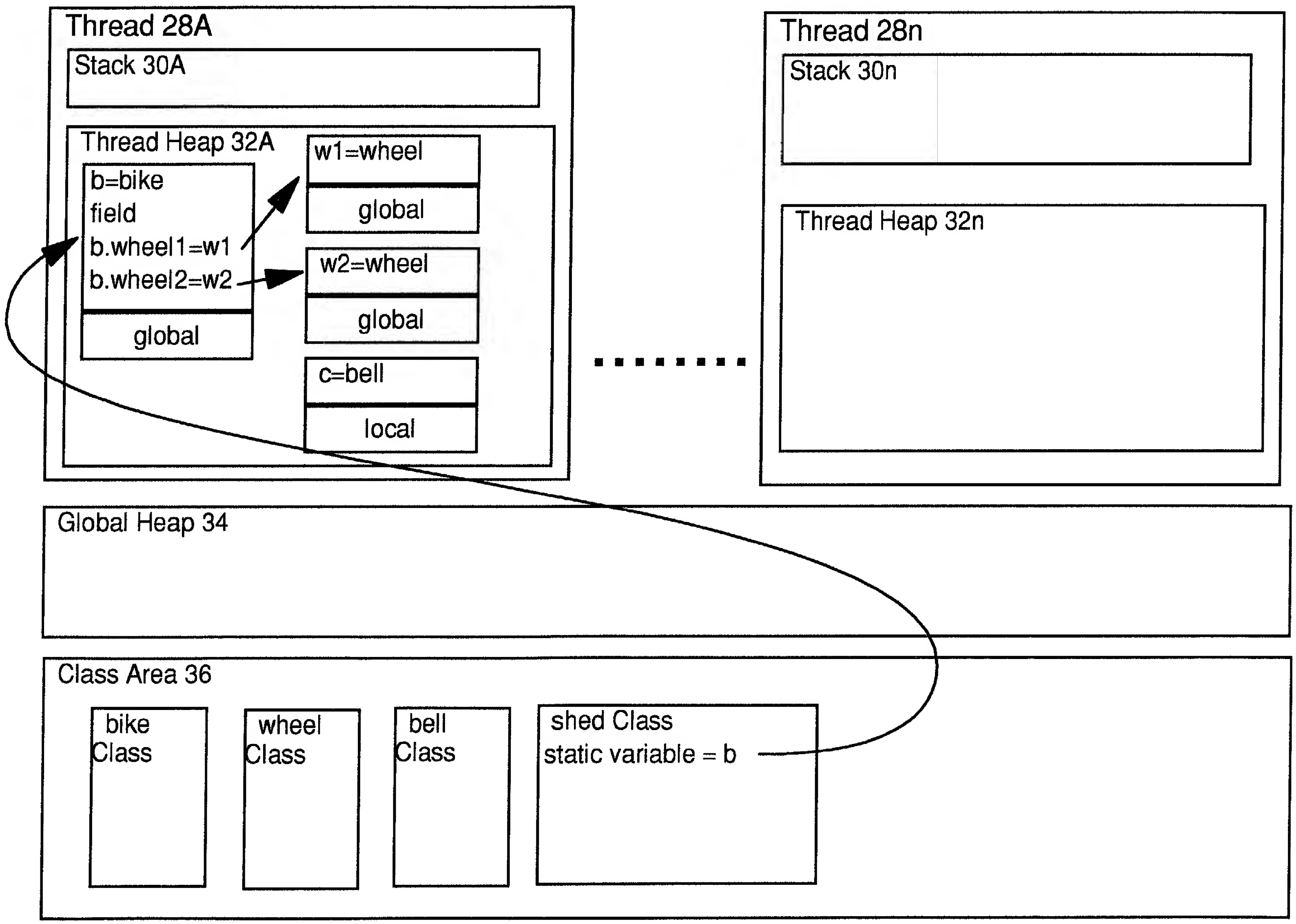

Thread 28A
Stack 30A
Thread Heap 32A
b=bike
field
b.wheel1=w1
b.wheel2=w2
global
w1=wheel
global
w2=wheel
global
c=bell
local
Thread 28n
Stack 30n
Thread Heap 32n
Global Heap 34
Class Area 36
bike
Class
wheel
Class
bell
Class
shed Class
static variable = b

Figure 6

DOCKET NO: UK998094

DECLARATION AND POWER OF ATTORNEY FOR PATENT APPLICATION

As a below named inventor, I hereby declare that:

My residence, post office address and citizenship are as stated below next to my name; I believe I am the original, first and sole inventor (if only one name is listed below) or an original, first and joint inventor (if plural names are listed below) of the subject matter which is claimed and for which a patent is sought on the invention entitled

VIRTUAL MACHINE MEMORY MANAGEMENT

the specification of which (check one)

__X__ is attached hereto.

______ was filed on
as Application Serial No.
and was amended on

I hereby state that I have reviewed and understand the contents of the above identified specification, including the claims, as amended by any amendment referred to above.

I acknowledge the duty to disclose information which is material to the patentability of this application in accordance with Title 37, CFR 1.56.

I hereby claim foreign priority benefits under Title 35, USC 119 of any foreign application(s) for patent or inventor's certificate listed below and have also identified below any foreign application for patent or inventor's certificate having a filing date before that of the application on which priority is claimed:

Prior Foreign Application(s):

Number	Country	Filing Date Day/Month/Year	Priority Claimed
9828298.1	United Kingdom	23 December 1998	__X__Yes _____No

I hereby claim the benefit under Title 35, USC 120 of any United States application(s) listed below and, insofar as the subject matter of each of the claims of this application is not disclosed in the prior United States application in the manner provided by the first paragraph of Title 35, USC 112, I acknowledge the duty to disclose information material to the patentability of this application as defined in Title 37, CFR 1.56 which occurred between the filing date of the prior application and the national or PCT international filing date of this application:

Prior U.S. Applications:

Serial No.	Filing Date	Status

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

POWER OF ATTORNEY: As a named inventor, I hereby appoint the following attorneys and/or agents to prosecute this application and transact all business in the Patent and Trademark Office connected therewith:

Manny W. Schecter, Reg. No. 31,722; William B. Porter, Reg. No. 33,135; Douglas W. Cameron, Reg. No. 31,596; Kevin M. Jordan, Reg. No. 40,277; Stephen C. Kaufman, Reg. No. 29,551; Richard M. Ludwin, Reg. No. 33,010; Daniel P. Morris, Reg. No, 32,053; Louis J. Percello, Reg. No. 33,206; Jay P. Sbrollini, Reg. No. 36,266; Stephen S. Strunck, Reg. No. 28,672; Robert P. Tassinari, Reg. No. 36,030; Robert M. Trepp, Reg. No. 25,933; Marc D. Schechter, Reg. No. 28,989; Louis P. Herzberg, Reg. No. 41,500.

Send all correspondence to:
Jay P. Sbrollini
IBM Corp, IP Law Dept,
T J Watson Research Center, PO Box 218,
Yorktown Heights, New York 10598.
Telephone: (914)945-2587

E.K. Elliot

FULL NAME OF SOLE OR FIRST INVENTOR: ~~Eliot~~ **Karl Kolodner**

INVENTOR'S SIGNATURE: Elliot Karl Kolodner DATE: May 12, 1999

RESIDENCE: 40 Hanna Street, Haifa ~~3817~~, Israel 34817 E.K.

CITIZENSHIP: Israel + U.S.A. E.K.

POST OFFICE ADDRESS: As above

FULL NAME OF SECOND INVENTOR: **Martin John Trotter**

INVENTOR'S SIGNATURE: Martin John Trotter DATE: 29th April 1999

RESIDENCE: 2 Broadgate Cottages, Potters Heron Close, Ampfield, Romsey, Hampshire SO51 9BX, United Kingdom

CITIZENSHIP: British

POST OFFICE ADDRESS: As above

FULL NAME OF THIRD INVENTOR:

INVENTOR'S SIGNATURE: ______________________ DATE: ____________

RESIDENCE:

CITIZENSHIP:

POST OFFICE ADDRESS: